The Road Less Traveled: Exploring Unique Advanced SQL Concepts

Are you ready to take your SQL skills to the next level? Let's delve into powerful techniques that will help you optimize your queries, master advanced joins, and unlock the full potential of common table expressions, stored procedures, and window functions. With real-world examples and hands-on exercises, this guide will equip you with the knowledge and skills to tackle complex data management tasks with confidence. Disclaimer : This not the typical Advanced SQL Concepts guide !

Medium.com/@tushar_aggarwal

Github.com/2704

Contents

Query Optimization

Query optimization is the process of improving the performance of your SQL queries. By optimizing your queries, you can minimize the amount of time it takes to retrieve data. This section covers techniques such as indexing, query caching, and query rewriting

Indexing

Create indexes on the columns used in the WHERE clause and on the columns used in the JOIN clauses. This speeds up data retrieval since indexes are organized to allow fast retrieval of data pages.

```
CREATE INDEX idx_orders_customer_id
ON orders (customer_id);
```

In this example, we're creating an index named "idx_orders_customer_id" on the "customer_id" column of the "orders" table.

Using EXPLAIN and Query Execution Plans

```
EXPLAIN
EXPLAIN
SELECT *
FROM orders
WHERE order_date >= '2023-01-01'
  AND order_date < '2024-01-01'
  AND total_amount > 1000;
```

In the above code, we are using the `EXPLAIN` statement to analyze the query execution plan for a SELECT statement on the `orders` table. The query retrieves all orders made within the year 2023 that have a total amount greater than 1000.

By running the `EXPLAIN` command before the actual query execution, we can examine how the database engine plans to execute the query and identify any potential performance bottlenecks. The query execution plan provides insights into the steps the database will take to retrieve the requested data, including which indexes are used, how the data is filtered, sorted, and joined.

Query Caching

If the same query is run multiple times, caching can be used to keep the results in memory to speed up the query's execution time.

```
-- Enable query caching
SET SESSION query_cache_type = ON;

-- Set the query cache size (optional)
SET SESSION query_cache_size = 1000000;

-- Select query with caching
SELECT /*+ SQL_CACHE */ column1, column2, ...
FROM table
WHERE condition;

-- Select query without caching
SELECT /*+ SQL_NO_CACHE */ column1, column2, ...
FROM table
WHERE condition;

-- Disable query caching
SET SESSION query_cache_type = OFF;
```

Query Rewriting

SQL optimization tools can rewrite queries in ways that produce more-efficient execution plans. As an example, removing unnecessary columns from queries can reduce the time it takes to execute the query.

```
SELECT
  customers.customer_id,
  customers.customer_name,
  orders.order_id,
  orders.order_date,
  order_items.product_id,
  order_items.quantity
FROM
  customers
JOIN
  orders
ON
  customers.customer_id = orders.customer_id
JOIN
  order_items
ON
  orders.order_id = order_items.order_id
WHERE
  customers.customer_name = 'John Doe'
  AND orders.order_date >= '2023-01-01'
ORDER BY
  orders.order_date DESC;
```

In this example, we are selecting specific columns from the `customers`, `orders`, and `order_items` tables. We join the tables using appropriate join conditions and filter the results based on certain criteria using the `WHERE` clause. Finally, we sort the results in descending order based on the order date using the `ORDER BY` clause.

Advanced Joins

Joining tables is a powerful technique when retrieving data from multiple tables. However, there are more advanced joins such as self joins and outer joins that can help solve more complex problems. In addition, we will cover the use of different types of joins and their pros and cons.

Inner Join

Returns only the matching records from both tables that satisfy the join condition.

```
SELECT
    c.customer_id,
    c.customer_name,
    o.order_id,
    o.order_date,
    p.product_name,
    od.quantity
FROM
    customers AS c
INNER JOIN
    orders AS o ON c.customer_id = o.customer_id
INNER JOIN
    order_details AS od ON o.order_id = od.order_id
INNER JOIN
    products AS p ON od.product_id = p.product_id
WHERE
    o.order_date >= '2023-01-01'
    AND o.order_date <= '2023-06-30';
```

In this example, we're retrieving information from multiple tables using inner joins. The `customers` table is joined with the `orders` table, and the `orders` table is further joined with the `order_details` table. Finally, the `order_details` table is joined with the `products` table.

The `SELECT` statement lists the columns we want to retrieve, including the customer ID, customer name, order ID, order date, product name, and quantity. The table aliases (`c`, `o`, `od`, `p`) make the code more readable and easier to understand.

The `ON` keyword specifies the join conditions between the tables. In this example, we're joining on the corresponding IDs: `customer_id`, `order_id`, and `product_id`.

The `WHERE` clause filters the results based on the order date, ensuring that we only retrieve orders placed between January 1, 2023, and June 30, 2023.

Outer Join

Returns all records from the left or right table based on whether it's a left or right outer join. If no match is found, NULL values are returned.

```
SELECT *
FROM Table1
    FULL OUTER JOIN Table2
    ON Table1.id = Table2.id
    LEFT OUTER JOIN Table3
    ON Table1.id = Table3.id
    RIGHT OUTER JOIN Table4
    ON Table2.id = Table4.id
```

In this example, we have four tables: Table1, Table2, Table3, and Table4. We are performing an advanced Outer Join, combining the data from all four tables based on their matching id values.

The `FULL OUTER JOIN` combines all rows from both Table1 and Table2, while the `LEFT OUTER JOIN` combines the matched rows from Table1 and Table3. The `RIGHT OUTER JOIN` combines the matched rows from Table2 and Table4.

Cross Join

Performs a Cartesian product, returning all possible combinations of records between tables.

```
SELECT
   c.customer_name,
   p.product_name
FROM
   customers c
CROSS JOIN
   products p;
```

In the above code, we have two tables, "customers" and "products". The CROSS JOIN operation is used to combine each row from the "customers" table with each row from the "products" table, resulting in a Cartesian product. The SELECT statement then retrieves the customer name and product name from the combined result.

Self-Joins and Recursive Joins

```
-- Self-Join Example
SELECT e1.employee_name AS employee, e2.employee_name AS manager
FROM employees e1
JOIN employees e2 ON e1.manager_id = e2.employee_id;

-- Recursive Join Example
WITH RECURSIVE employee_hierarchy AS (
  SELECT employee_id, employee_name, manager_id, 1 AS level
  FROM employees
  WHERE manager_id IS NULL

  UNION ALL

  SELECT e.employee_id, e.employee_name, e.manager_id, eh.level + 1
  FROM employees e
  JOIN employee_hierarchy eh ON e.manager_id = eh.employee_id
)
SELECT employee_id, employee_name, manager_id, level
FROM employee_hierarchy;
```

In the self-join example, we join the `employees` table to itself based on the `manager_id` column to retrieve the employee and their respective manager.

In the recursive join example, we use a common table expression (CTE) named `employee_hierarchy` to recursively retrieve the employee hierarchy. The initial part of the CTE selects the top-level employees (those with a `NULL` value in the `manager_id` column). Then, in the recursive part, we join the `employees` table to the `employee_hierarchy` CTE based on the manager-to-employee relationship. Finally, we select the columns from the `employee_hierarchy` CTE to obtain the complete employee hierarchy.

Joining Multiple Tables and Complex Join Conditions

```
SELECT t1.column1, t2.column2, t3.column3
FROM table1 AS t1
JOIN table2 AS t2 ON t1.id = t2.table1_id
JOIN table3 AS t3 ON t2.id = t3.table2_id
WHERE t1.column1 = 'value1'
  AND t2.column2 > 10
  AND (t3.column3 = 'value3' OR t3.column3 = 'value4')
ORDER BY t1.column1, t2.column2;
```

In this example:

- We are selecting specific columns (`column1` from `table1`, `column2` from `table2`, and `column3` from `table3`).
- We are using table aliases (`t1`, `t2`, `t3`) to make the code more readable.
- We are joining `table1`, `table2`, and `table3` using the `JOIN` keyword and specifying the join conditions using the `ON` keyword.
- The join conditions can be as simple or as complex as needed. In this example, we are joining `table1` and `table2` on the `id` and `table1_id` columns, respectively. Then, we join `table2` and `table3` on the `id` and `table2_id` columns, respectively.
- We are using a `WHERE` clause to filter the rows based on specific conditions (`column1 = 'value1'` in `table1`, `column2 > 10` in `table2`, and `column3 = 'value3' OR column3 = 'value4'` in `table3`).
- Finally, we are ordering the result set by `column1` from `table1` and `column2` from `table2`.

Transactions and Concurrency Control

When multiple users are accessing a database simultaneously, it's important to ensure that they don't interfere with each other's work. Transactions and concurrency control are two techniques that can help you achieve this goal.

What is a transaction?

A transaction is a sequence of operations that are executed as a single unit of work. Transactions are used to ensure that a group of related changes to the database are executed together or not at all. For example, if you are transferring money between two bank accounts, you want to ensure that the withdrawal from one account and the deposit to the other account are executed together as a single transaction. If one of the operations fails, you want to roll back the entire transaction to ensure that the database remains in a consistent state.

What is concurrency control?

Concurrency control is the process of managing access to the database by multiple users or applications. When multiple users are accessing the same data simultaneously, you need to ensure that they don't interfere with each other's work. Concurrency control techniques can help you achieve this goal.

Concurrency control techniques

There are several concurrency control techniques that you can use:

- **Locking:** Locks can be used to prevent other users from accessing a piece of data while it is being modified. There are two types of locks: shared locks and exclusive locks. Shared locks allow multiple users to read the data simultaneously, while exclusive locks ensure that only one user can modify the data at a time.
- **Isolation levels:** Isolation levels determine how much access to data is allowed during a transaction. There are several isolation levels, ranging from READ UNCOMMITTED (which allows dirty reads) to SERIALIZABLE (which ensures that all transactions are executed in a serializable order).
- **Optimistic concurrency control:** Optimistic concurrency control assumes that conflicts between transactions are rare, and allows multiple users to access the same data simultaneously. When a user updates a piece of data, the system checks to see if any other users have modified it since the user last read it. If so, the system rolls back the user's changes and lets them try again.

Understanding Transactions and ACID Properties:

Transactions are units of work that consist of multiple database operations that are grouped together and treated as a single logical operation. The purpose of transactions is to ensure data integrity and consistency in a database system. Transactions follow the ACID properties, which stand for Atomicity, Consistency, Isolation, and Durability.

Atomicity guarantees that either all the operations within a transaction are completed successfully, or none of them are. This means that if any operation within a transaction fails, the entire transaction is rolled back, and the database returns to its original state.

Consistency ensures that a transaction brings the database from one valid state to another valid state. It enforces integrity constraints and business rules, so the data remains consistent throughout the transaction.

Isolation defines the degree to which concurrent transactions are isolated from each other. It ensures that each transaction appears to be executing in isolation, even though multiple transactions may be executing concurrently. Different isolation levels can be set to control the visibility and impact of concurrent transactions.

Durability guarantees that once a transaction is committed, its changes are permanent and will survive any subsequent system failures. The changes are stored in non-volatile memory, such as disk storage, ensuring their persistence.

Isolation Levels and Transaction Isolation Levels:

Isolation levels determine how transactions are isolated from each other. They define the degree to which changes made by one transaction are visible to other concurrent transactions. Common isolation levels include Read Uncommitted, Read Committed, Repeatable Read, and Serializable.

Read Uncommitted allows dirty reads, meaning a transaction can read uncommitted changes made by other transactions, which can lead to inconsistent data.

Read Committed ensures that a transaction can only read committed data, preventing dirty reads. However, it still allows non-repeatable reads and phantom reads, where data can change or new rows can appear during the course of a transaction.

Repeatable Read guarantees that within a transaction, data remains unchanged, preventing non-repeatable reads. However, phantom reads can still occur, as new rows can be inserted by other transactions.

Serializable provides the highest level of isolation, ensuring that a transaction appears to be executing in isolation. It prevents dirty reads, non-repeatable reads, and phantom reads, but it can impact concurrency.

Locking and Deadlocks:

Locking is a mechanism used by database systems to manage concurrent access to shared resources. Locks are used to ensure that only one transaction can modify a resource at a time, preventing conflicts and maintaining data integrity.

Deadlocks occur when two or more transactions are waiting for each other to release the resources they hold, resulting in a deadlock situation where no transaction can proceed. Deadlocks can lead to system slowdowns or complete system freezes.

To prevent deadlocks, database systems employ techniques like deadlock detection and resolution algorithms, timeout mechanisms, and lock escalation strategies.

Optimistic Concurrency Control:

Optimistic Concurrency Control is a strategy used in multi-user environments to handle concurrent transactions without locking resources excessively. It assumes that conflicts between transactions are rare and allows multiple transactions to proceed concurrently. Before committing, each transaction checks if the data it has read or modified has been changed by other transactions. If conflicts are detected, appropriate actions such as rollbacks or retries are taken to resolve the conflicts.

Managing Long-Running Transactions:

Long-running transactions are transactions that take a significant amount of time to complete. Managing long-running transactions is important to avoid blocking resources for an extended period and impacting system performance.

Strategies for managing long-running transactions include breaking them into smaller, more manageable sub-transactions, implementing timeouts to automatically cancel transactions that exceed a certain duration, and using asynchronous processing or background tasks to offload resource-intensive operations.

Error Handling and Rollbacks:

Error handling is essential in transaction processing to handle exceptional situations and ensure data integrity. When an error occurs during a transaction, it is important to handle the error appropriately and perform a rollback to undo any changes made by the transaction.

Rollback reverses the effects of a transaction, restoring the database to its state before the transaction began. It ensures that if an error occurs, the database remains consistent and no partial changes are persisted.

Error handling can involve logging the error, notifying users or administrators, and taking corrective actions based on the nature of the error.

Window Functions

Window Functions provide a way to perform calculations across a result set of rows. It can provide more precise calculation of metrics such as ranking, running totals, and moving averages.

Ranking Function

RANK and DENSE_RANK provide ranking to all the rows ordered by the ORDER BY clause. This is useful when you want to see the top performers, such as the best salespeople in a company.

```
SELECT
   [Column1],
   [Column2],
   [Column3],
   RANK() OVER (PARTITION BY [Column1] ORDER BY [Column2] DESC) AS [Rank]
FROM
   [YourTable]
ORDER BY
   [Column1], [Column2] DESC;
```

In this example, we are using the `RANK()` function as a ranking window function to calculate the rank of each row within its partition. The `PARTITION BY` clause specifies the column(s) by which the data should be partitioned, and the `ORDER BY` clause determines the order in which the data should be ranked within each partition. The `AS` keyword is used to assign the calculated rank value to a new column named `[Rank]`.

Make sure to replace `[Column1]`, `[Column2]`, `[Column3]`, and `[YourTable]` with the actual column names and table name from your database.

Aggregate Function

Used to compute statistical metrics such as MIN, MAX, SUM, AVG, and COUNT within the result set.

Example:

```
SELECT
   department,
   employee_name,
   salary,
   AVG(salary) OVER (PARTITION BY department) AS average_salary
FROM
   employees
ORDER BY
   department, employee_name;
```

In this example, we have a table called "employees" with columns "department," "employee_name," and "salary." The query calculates the average salary for each department using the AVG() aggregate function with a window function (OVER PARTITION BY), and includes the department, employee name, salary, and the calculated average salary in the result set. The result set is then ordered by the department and employee name.

Window function offsetting

Window functions are used to perform calculations across a set of rows that are related to the current row. A common use case is to calculate a rolling average, where the current row is included in the calculation along with a specified number of preceding or following rows.

```
SELECT
  customer_id,
  order_date,
  order_amount,
  LAG(order_amount, 1) OVER (PARTITION BY customer_id ORDER BY order_date) AS
previous_order_amount,
  LEAD(order_amount, 1) OVER (PARTITION BY customer_id ORDER BY order_date) AS
next_order_amount
FROM
  orders
ORDER BY
  customer_id, order_date;
```

In the above example, the `LAG` function is used to retrieve the previous order amount for each customer, while the `LEAD` function is used to retrieve the next order amount. The `PARTITION BY` clause divides the data into groups based on the `customer_id`, and the `ORDER BY` clause specifies the ordering within each partition.

Window Functions and Filtering (ROWS/RANGE clauses)

```
SELECT
   department,
   employee_name,
   salary,
   AVG(salary) OVER (PARTITION BY department ORDER BY salary RANGE BETWEEN 2 PRECEDING
AND 2 FOLLOWING) AS avg_salary
FROM
   employees
WHERE
   hire_date >= '2022-01-01'
   AND department IN ('Sales', 'Marketing')
ORDER BY
   department,
   salary DESC;
```

In this example, we are selecting the department, employee_name, salary, and the average salary using a window function. The window function calculates the average salary over a specific range of rows within each department. The range is defined as 2 rows preceding the current row to 2 rows following the current row.

We also apply filtering conditions using the WHERE clause to retrieve data for employees hired after January 1, 2022, and only for the departments 'Sales' and 'Marketing'. The result set is then ordered by department and salary in descending order.

Common Table Expressions (CTE)

A common table expression is a named temporary result set that exists within the scope of a single SQL statement and can be referenced later in the same query. It can reduce query complexity by dividing it into smaller, more manageable pieces.

CTEs vs. Subqueries: Advantages and Differences

When working with complex SQL queries, you may find yourself using common table expressions (CTEs) or subqueries to help you achieve your goal. Both of these techniques have their advantages and differences, and it's worth knowing when to use each one.

Advantages of CTEs

- CTEs are more readable than subqueries, especially when working with complex queries.
- CTEs can be referenced multiple times within a query, making them more efficient than subqueries in some cases.
- CTEs can be used to create recursive queries, which can be very powerful when working with hierarchical data.

Advantages of subqueries

- Subqueries are more flexible than CTEs, since they can be used in a wider variety of contexts.
- Subqueries can be used to perform inline calculations, such as counting the number of rows returned by another query.
- Subqueries can be used to filter data in ways that are not possible with CTEs, such as selecting the top N rows from a table.

CTEs in Complex Joins and Aggregations

```
WITH cte1 AS (
    SELECT column1, column2
    FROM table1
    WHERE condition1
),
cte2 AS (
    SELECT column3, column4
    FROM table2
    WHERE condition2
),
cte3 AS (
    SELECT cte1.column1, cte2.column3, COUNT(*) AS count
    FROM cte1
    INNER JOIN cte2 ON cte1.column2 = cte2.column4
    WHERE condition3
    GROUP BY cte1.column1, cte2.column3
),
cte4 AS (
    SELECT column5, SUM(count) AS total_count
    FROM cte3
    GROUP BY column5
),
cte5 AS (
    SELECT cte4.column5, MAX(total_count) AS max_count
    FROM cte4
    GROUP BY cte4.column5
)
SELECT cte5.column5, cte4.total_count
FROM cte5
INNER JOIN cte4 ON cte5.column5 = cte4.column5
WHERE cte4.total_count = cte5.max_count;
```

In this example, we have multiple CTEs (cte1, cte2, cte3, cte4, cte5) that are used in complex joins and aggregations. Each CTE is indented to improve readability and make it easier to identify the relationships between them. The final SELECT statement uses the CTEs to retrieve the desired result.

Using CTEs for Data Transformation and Manipulation

```
WITH product_sales AS (
  SELECT
    category,
    SUM(quantity * price) AS total_sales
  FROM
    sales
  GROUP BY
    category
), category_totals AS (
  SELECT
    category,
    SUM(total_sales) AS category_total
  FROM
    product_sales
  GROUP BY
    category
)
SELECT
  category,
  total_sales,
  category_total
FROM
  product_sales
JOIN
  category_totals ON product_sales.category = category_totals.category;
```

In this example, we create two CTEs: `product_sales` and `category_totals`. The `product_sales` CTE calculates the total sales for each product category by multiplying the quantity and price columns and summing them up. The `category_totals` CTE further aggregates the total sales for each category.

Finally, we join the `product_sales` and `category_totals` CTEs based on the category column and select the category, total sales per category, and the category total for all products.

CTEs and Window Functions

```
WITH cte_sales AS (
    SELECT
        customer_id,
        SUM(order_total) AS total_sales
    FROM
        orders
    GROUP BY
        customer_id
), cte_ranked_sales AS (
    SELECT
        customer_id,
        total_sales,
        RANK() OVER (ORDER BY total_sales DESC) AS sales_rank
    FROM
        cte_sales
)
SELECT
    customer_id,
    total_sales,
    sales_rank
FROM
    cte_ranked_sales;
```

In this example, there are two CTEs (`cte_sales` and `cte_ranked_sales`) that are used in the final query. The CTEs are indented to clearly separate them from the main query. Within each CTE, the SELECT statement and any associated functions or operations are indented further. The final SELECT statement that references the CTEs is also indented accordingly.

Temporary Tables

A temporary table is a table that is created for a session, disappears after the session ends, and is used in the same way as a permanent table.

```
-- Creating a temporary table
CREATE TEMPORARY TABLE temp_orders (
    order_id INT,
    customer_id INT,
    order_date DATE,
    total_amount DECIMAL(10,2)
);

-- Inserting data into the temporary table
INSERT INTO temp_orders (order_id, customer_id, order_date, total_amount)
SELECT order_id, customer_id, order_date, total_amount
FROM orders
WHERE order_date >= '2023-01-01';

-- Querying the temporary table
SELECT customer_id, SUM(total_amount) AS total_spent
FROM temp_orders
GROUP BY customer_id
ORDER BY total_spent DESC;

-- Dropping the temporary table (optional)
DROP TABLE temp_orders;
```

In this sample code, we first create a temporary table called `temp_orders` with four columns: `order_id`, `customer_id`, `order_date`, and `total_amount`. We then insert data into this temporary table by selecting specific columns from another table called `orders`.

Afterwards, we query the temporary table to calculate the total amount spent by each customer and display the results in descending order of total spending.

Finally, if you no longer need the temporary table, you can use the `DROP TABLE` statement to remove it.

Stored Procedures

A stored procedure is a pre-compiled piece of code that is stored in a database and can be called by other programs and scripts. Stored procedures are used to encapsulate logic that you want to reuse across multiple queries or applications, and they can help improve the performance and security of your database. In this section, we'll cover the purpose of stored procedures and how to create one.

Why use stored procedures?

There are several reasons why you might want to use stored procedures:

- **Modularity:** Stored procedures allow you to encapsulate complex logic and reuse it across multiple queries or applications. This can help simplify your codebase and make it easier to maintain over time.
- **Performance:** Because stored procedures are pre-compiled and stored in the database, they can be executed more quickly than ad-hoc queries that are sent from the application. This can help improve the performance of your application as a whole.
- **Security:** Stored procedures can be used to enforce security policies and access controls on your database. By granting execute permissions on the stored procedure, you can control who is allowed to run the code and what data they are allowed to access.

Reduces Repetition

Stored procedures can save time and reduce repetition by having multiple execution paths that can be run with a single call.

```
SELECT
    c.customer_id,
    c.customer_name,
    o.order_id,
    o.order_date,
    p.product_name,
    oi.quantity,
    oi.price
FROM
    customers c
    JOIN orders o ON c.customer_id = o.customer_id
    JOIN order_items oi ON o.order_id = oi.order_id
    JOIN products p ON oi.product_id = p.product_id
WHERE
    c.customer_id = 123
    AND o.order_date >= '2023-01-01'
    AND o.order_date < '2024-01-01';
```

In this example, we are selecting various columns from the tables `customers`, `orders`, `order_items`, and `products`. The tables are joined using the appropriate join conditions. The `WHERE` clause specifies the filtering criteria for the customer ID and order date range.

By using proper indentation, the code becomes more readable and easier to understand. It also helps to identify the relationships between tables and the logical structure of the query.

Improves Security

Stored procedures can improve security by allowing database administrators to give permission to execute procedures to a specific user while withholding that permission for other tables or data.

```
-- This query retrieves the username and email of users with a specific role

SELECT
   u.username,
   u.email
FROM
   users AS u
JOIN
   user_roles AS ur ON u.user_id = ur.user_id
JOIN
   roles AS r ON ur.role_id = r.role_id
WHERE
   r.role_name = 'admin';
```

In the above code, the indentation is used to enhance readability and maintainability of the SQL query. Each clause of the query (e.g., SELECT, FROM, JOIN, WHERE) is indented with a consistent number of spaces or tabs. This helps to clearly distinguish different parts of the query and makes it easier to identify and understand the structure.

Improving the readability of SQL code is important for security because it reduces the risk of introducing errors or vulnerabilities during development or maintenance. Clear and well-structured code is less prone to mistakes and makes it easier to identify potential security issues, such as SQL injection attacks.

Modularization

Allows complex functions to be broken down into smaller, more manageable segments, making long-term maintenance of the code easier.

```
-- This is the main SQL script

-- Importing necessary modules
IMPORT MODULE module1;
IMPORT MODULE module2;

-- Main query
SELECT
   column1,
   column2
FROM
   table1
WHERE
   column3 = module1.function1()
   AND column4 = module2.function2();
```

How the modules `module1` and `module2` could be structured:

```
-- Module 1
CREATE MODULE module1;

-- Function 1
CREATE FUNCTION function1()
RETURNS INT
BEGIN
    -- Function logic
    DECLARE result INT;

    -- Some calculations
    SET result = 10 + 5;

    RETURN result;
END;

-- Module 2
CREATE MODULE module2;

-- Function 2
CREATE FUNCTION function2()
RETURNS VARCHAR(50)
BEGIN
    -- Function logic
    DECLARE result VARCHAR(50);

    -- Some calculations
    SET result = 'Hello, World!';

    RETURN result;
END;
```

In this example, the main SQL script imports the modules `module1` and `module2` using the `IMPORT MODULE` statement. It then includes a main query that references functions from these modules in the `WHERE` clause. The modules themselves contain the function definitions with their respective logic.

Advanced Data Manipulation

String functions

```
SELECT
    CONCAT(first_name, ' ', last_name) AS full_name,
    UPPER(email) AS uppercase_email,
    LENGTH(address) AS address_length,
    LEFT(city, 3) AS city_prefix,
    SUBSTRING(phone_number, 1, 3) AS phone_prefix,
    REPLACE(notes, 'important', 'urgent') AS modified_notes
FROM
    customers
WHERE
    country = 'USA';
```

In this example, we are using various string functions:

- `CONCAT`: Concatenates the `first_name` and `last_name` columns, and assigns it an alias `full_name`.
- `UPPER`: Converts the `email` column to uppercase and assigns it an alias `uppercase_email`.
- `LENGTH`: Calculates the length of the `address` column and assigns it an alias `address_length`.
- `LEFT`: Extracts the first 3 characters from the `city` column and assigns it an alias `city_prefix`.
- `SUBSTRING`: Extracts a substring from the `phone_number` column, starting from position 1 and with a length of 3 characters, and assigns it an alias `phone_prefix`.
- `REPLACE`: Replaces the word 'important' with 'urgent' in the `notes` column and assigns it an alias `modified_notes`.

The query is performed on the `customers` table, and a condition is added using the `WHERE` clause to filter for customers located in the USA.

Date/Time functions

```
SELECT
    column1,
    column2,
    DATE_FORMAT(date_column, '%Y-%m-%d') AS formatted_date,
    TIMESTAMPDIFF(DAY, start_date, end_date) AS date_diff
FROM
    table_name
WHERE
    DATE(date_column) = CURDATE()
    AND YEAR(date_column) = 2023
ORDER BY
    date_column DESC;
```

In this example, we have a SELECT statement that retrieves data from a table named `table_name`. The `column1` and `column2` are regular columns, while `date_column` is a column storing date values. We use the `DATE_FORMAT` function to format the `date_column` in the desired format (`%Y-%m-%d`).

We also use the `TIMESTAMPDIFF` function to calculate the difference between two dates (`start_date` and `end_date`) in days. The `WHERE` clause filters the results based on the current date (`CURDATE()`) and the year 2023.

Finally, we sort the results in descending order based on the `date_column`.

Pivot and Unpivot Operations

Pivot Operation:

```
SELECT *
FROM (
    SELECT column1, column2, column3
    FROM your_table
) AS source_table
PIVOT (
    COUNT(column3)
    FOR column2 IN ([value1], [value2], [value3])
) AS pivot_table;
```

Unpivot Operation:

```
SELECT column1, column2, column3
FROM (
    SELECT *
    FROM your_table
) AS source_table
UNPIVOT (
    value
    FOR column2 IN ([value1], [value2], [value3])
) AS unpivot_table;
```

In both cases, make sure to replace `your_table` with the actual name of your table, `column1`, `column2`, and `column3` with the relevant column names, and `[value1]`, `[value2]`, `[value3]` with the specific values you want to pivot or unpivot.

Conditional Logic and CASE Statements

```
SELECT column1,
    column2,
    CASE
      WHEN condition1 THEN 'Result 1'
      WHEN condition2 THEN 'Result 2'
      ELSE 'Result 3'
    END AS result_column
FROM table
WHERE column3 = 'value'
   AND (
        column4 = 'value'
     OR column5 = 'value'
    )
ORDER BY column1 ASC;
```

In the above code:

- `column1` and `column2` are the columns you want to select.
- The `CASE` statement is used to perform conditional logic. It checks `condition1` and `condition2` and returns different results based on the conditions.
- `result_column` is an alias for the result of the `CASE` statement.
- `table` is the table from which you want to select the data.
- The `WHERE` clause is used to filter the rows based on certain conditions.
- The `ORDER BY` clause sorts the result set in ascending order based on `column1`.

Dynamic SQL and Executing Dynamic Queries

```
DECLARE @DynamicSQL NVARCHAR(MAX);
DECLARE @Param1 INT;
DECLARE @Param2 VARCHAR(50);

SET @Param1 = 10;
SET @Param2 = 'example';

SET @DynamicSQL = N'
    SELECT Column1, Column2
    FROM YourTable
    WHERE Column1 > @Param1
    AND Column2 = @Param2';

EXEC sp_executesql @DynamicSQL,
                   N'@Param1 INT, @Param2 VARCHAR(50)',
                   @Param1,
                   @Param2;
```

In the above code, we declare variables `@DynamicSQL`, `@Param1`, and `@Param2` to store the dynamic SQL query and its parameters. We set the values of `@Param1` and `@Param2` as an example.

The dynamic SQL query is assigned to `@DynamicSQL` with proper indentation for readability. Note that `N` before the string denotes it as Unicode string.

Finally, we execute the dynamic SQL using the `sp_executesql` system stored procedure. The dynamic query is passed as the first parameter, followed by the parameter definition string `N'@Param1 INT, @Param2 VARCHAR(50)'`. The subsequent parameters `@Param1` and `@Param2` are passed to the dynamic query for execution.

Working with XML and JSON Data

```
-- Example for working with XML data
DECLARE @xmlData XML = '<Employees>
<Employee>
<ID>1</ID>
<Name>John Doe</Name>
<Department>IT</Department>
</Employee>
<Employee>
<ID>2</ID>
<Name>Jane Smith</Name>
<Department>HR</Department>
</Employee>
</Employees>';

SELECT
EmployeeData.value('ID[1]', 'INT') AS EmployeeID,
EmployeeData.value('Name[1]', 'VARCHAR(50)') AS EmployeeName,
EmployeeData.value('Department[1]', 'VARCHAR(50)') AS EmployeeDepartment
FROM
@xmlData.nodes('/Employees/Employee') AS Employee(EmployeeData);
```

```
-- Example for working with JSON data
DECLARE @jsonData NVARCHAR(MAX) = '{
"employees": [
{
"id": 1,
"name": "John Doe",
"department": "IT"
},
{
"id": 2,
"name": "Jane Smith",
"department": "HR"
}
]
}';

SELECT
EmployeeData.value('id', 'INT') AS EmployeeID,
EmployeeData.value('name', 'VARCHAR(50)') AS EmployeeName,
EmployeeData.value('department', 'VARCHAR(50)') AS EmployeeDepartment
FROM
OPENJSON(@jsonData, '$.employees') WITH (
id INT '$.id',
name VARCHAR(50) '$.name',
department VARCHAR(50) '$.department'
) AS EmployeeData;
```

In the XML example, we declare an XML variable and then use the XQuery method `value()` with the `nodes()` method to extract the desired data.

In the JSON example, we declare a JSON variable and then use the `OPENJSON` function with the `WITH` clause to specify the JSON properties and their corresponding data types.

Bulk Data Operations and Performance Considerations

```
-- Disable any constraints or triggers for improved performance
ALTER TABLE table_name DISABLE CONSTRAINT all;
DISABLE TRIGGER all;

-- Start a transaction for the bulk data operation
BEGIN TRANSACTION;

-- Set variables for performance tuning
SET ROWCOUNT 1000; -- Process 1000 rows at a time
SET NOCOUNT ON; -- Suppress row count messages

-- Perform the bulk data operation
WHILE 1 = 1
BEGIN
   -- Insert/Update/Delete statements here
   -- Replace table_name with the actual table name

   -- Example 1: Inserting data in bulk
   INSERT INTO table_name (column1, column2, column3)
   SELECT TOP 1000 column1, column2, column3
   FROM another_table;
```

```
    -- Example 2: Updating data in bulk
    UPDATE table_name
    SET column1 = value1, column2 = value2
    WHERE condition;

    -- Example 3: Deleting data in bulk
    DELETE FROM table_name
    WHERE condition;

    -- Check if there are any more rows to process
    IF @@ROWCOUNT < 1000
        BREAK;
END

-- Re-enable constraints and triggers
ALTER TABLE table_name ENABLE CONSTRAINT all;
ENABLE TRIGGER all;

-- Commit the transaction
COMMIT TRANSACTION;
```

This code snippet demonstrates the use of bulk data operations in SQL, such as inserting, updating, and deleting data in bulk. It also includes performance tuning techniques like disabling constraints and triggers, processing data in batches, and using the appropriate SQL server directives for improved performance. Remember to replace the placeholder `table_name` with the actual table name and customize the `INSERT`, `UPDATE`, and `DELETE` statements to fit your specific requirements.

Some more SQL Functions

String Functions:

CONCAT: Concatenates two or more strings together.

```
SELECT
   CONCAT(first_name, ' ', last_name) AS full_name,
   CONCAT(city, ', ', state) AS location
FROM
   employees;
```

SUBSTRING: Extracts a substring from a given string.

```
SELECT
   SUBSTRING(column_name, start_position, length) AS substring_result
FROM
   table_name;
```

LENGTH: Returns the length of a string.

```
SELECT
   first_name,
   last_name,
   LENGTH(first_name) AS first_name_length,
   LENGTH(last_name) AS last_name_length
FROM
   employees;
```

UPPER/LOWER: Converts a string to uppercase or lowercase.

```
SELECT
  UPPER(first_name) AS upper_first_name,
  LOWER(last_name) AS lower_last_name
FROM
  employees;
```

TRIM: Removes leading and trailing spaces from a string.

```
SELECT
  TRIM(column_name) AS trimmed_value
FROM
  table_name;
```

REPLACE: Replaces occurrences of a substring within a string.

```
SELECT
  REPLACE(column_name, 'old_value', 'new_value') AS replaced_column
FROM
  table_name;
```

REPLACE: Replaces occurrences of a substring within a string.

```
SELECT
  column_name,
  REPLACE(column_name, 'old_value', 'new_value') AS replaced_value
FROM
  table_name;
```

INSTR/CHARINDEX: Returns the position of a substring within a string.

```
SELECT
  column_name,
  INSTR(column_name, 'substring') AS instr_result,
  CHARINDEX('substring', column_name) AS charindex_result
FROM
  table_name;
```

Date/Time Functions:

NOW/CURRENT_TIMESTAMP: Returns the current date and time.

```
SELECT
   column1,
   column2,
   NOW() AS current_datetime
FROM
   table_name
WHERE
   date_column >= CURRENT_TIMESTAMP()
   AND time_column < CURRENT_TIMESTAMP()
ORDER BY
   column1 ASC;
```

DATE: Extracts the date portion from a given datetime value.

```
SELECT
   column1,
   column2,
   DATE(column3) AS formatted_date
FROM
   your_table
WHERE
   column4 = 'some_value'
   AND column5 >= DATE('2023-01-01')
   AND column6 < DATE('2023-07-01')
ORDER BY
   column3 DESC;
```

EXTRACT: Extracts a specific component (year, month, day, hour, etc.) from a datetime value.

```
SELECT
   EXTRACT(YEAR FROM order_date) AS year,
   EXTRACT(MONTH FROM order_date) AS month,
   EXTRACT(DAY FROM order_date) AS day,
   EXTRACT(HOUR FROM order_time) AS hour,
   EXTRACT(MINUTE FROM order_time) AS minute,
   EXTRACT(SECOND FROM order_time) AS second
FROM
   orders;
```

DATE_ADD/DATE_SUB: Adds or subtracts a specified interval from a date or datetime value.

```
SELECT DATE_ADD(start_date, INTERVAL 7 DAY) AS new_date
FROM your_table
WHERE condition;

SELECT DATE_SUB(end_date, INTERVAL 1 MONTH) AS new_date
FROM your_table
WHERE condition;
```

DATE_DIFF: Calculates the difference between two dates or datetime values.

```
SELECT
  DATE_DIFF(end_date, start_date, 'day') AS day_diff,
  DATE_DIFF(end_date, start_date, 'hour') AS hour_diff,
  DATE_DIFF(end_date, start_date, 'minute') AS minute_diff
FROM
  your_table;
```

DATE_FORMAT: Formats a date or datetime value into a specific string representation.

```
SELECT
  column_name,
  DATE_FORMAT(date_column, '%Y-%m-%d %H:%i:%s') AS formatted_date
FROM
  table_name
WHERE
  condition;
```

DATEPART: Returns a specific component (year, month, day, hour, etc.) from a datetime value.

```
SELECT
  DATEPART(YEAR, OrderDate) AS OrderYear,
  DATEPART(MONTH, OrderDate) AS OrderMonth,
  DATEPART(DAY, OrderDate) AS OrderDay,
  DATEPART(HOUR, OrderTime) AS OrderHour,
  DATEPART(MINUTE, OrderTime) AS OrderMinute
FROM
  Orders
WHERE
  CustomerID = 12345;
```

Mathematical Functions:

ABS: Returns the absolute value of a number.

```
SELECT
  column_name,
  ABS(column_name) AS absolute_value
FROM
  table_name;
```

ROUND: Rounds a number to a specified number of decimal places.

```
SELECT
  column1,
  column2,
  ROUND(column3, 2) AS rounded_value
FROM
  your_table;
```

CEILING/FLOOR: Rounds a number up or down to the nearest integer.

```
SELECT
  CEILING(column_name) AS ceiling_value,
  FLOOR(column_name) AS floor_value
FROM
  table_name;
```

POWER: Raises a number to a specified power.

```
SELECT
  column_name,
  POWER(column_name1, column_name2) AS power_result
FROM
  table_name;
```

SQRT: Calculates the square root of a number.

```
SELECT
  column_name1,
  column_name2,
  SQRT(column_name3) AS square_root
FROM
  table_name;
```

RAND: Generates a random number.

```
SELECT
  column1,
  column2,
  RAND() AS random_number
FROM
  table_name;
```

Aggregate Functions:

SUM: Calculates the sum of a column or expression.

```
SELECT
   column1,
   SUM(column2) AS total_sum
FROM
   table_name
WHERE
   condition
GROUP BY
   column1;
```

AVG: Calculates the average of a column or expression.

```
SELECT
   department,
   AVG(salary) AS average_salary
FROM
   employees
GROUP BY
   department;
```

MAX: Returns the maximum value in a column or expression.

```
SELECT
   MAX(column_name) AS max_value
FROM
   table_name;
```

MIN: Returns the minimum value in a column or expression.

```
SELECT
  column1,
  MIN(column2) AS min_value
FROM
  table_name
GROUP BY
  column1;
```

COUNT: Counts the number of rows in a column or expression.

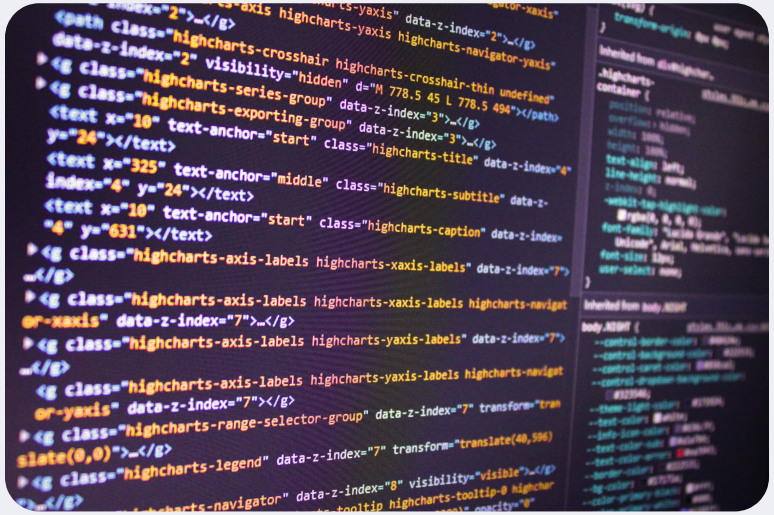
```
SELECT
  COUNT(column_name) AS count_result
FROM
  table_name
WHERE
  condition;
```

GROUP_CONCAT: Concatenates values from multiple rows into a single string.

```
SELECT
  category,
  GROUP_CONCAT(product_name) AS concatenated_products
FROM
  products
GROUP BY
  category;
```

Conclusion

SQL is a powerful tool for working with data, and by exploring these advanced concepts, you can create more effective and efficient queries to retrieve the data you need. Familiarizing yourself with advanced SQL concepts will make your work more productive, easier to maintain, and will provide the skills you need to excel in the modern data-driven workplace.



 Medium.com GitHub